A Survey of Formal Refinement Calculi

Albert Y. C. Lai

November, 2001

1 Introduction

A refinement calculus is essentially a formalization of the idea of stepwise refinement. In a refinement calculus, programs and specifications are unified and formalized as mathematical objects, and refinement is formalized as a relation between such objects. Program development in a refinement calculus is a process of top-down stepwise refinement: begin with a specification, refine each specification by a combination of new, more concrete specifications, until all specifications reach the level of programs. Each refinement is justified using the given formalization.

This paper surveys the area of refinement calculi. Section 2 summarizes the literature in this area. Section 3 introduces notations used in the rest of this paper. Section 4 describes and illustrates two refinement calculi in detail. Section 5 discusses the differences between the two refinement calculi described in Section 4 in several aspects. Section 6 identifies some current questions in refinement calculi.

2 Bibliography

This section summarizes the literature in the area of refinement calculi. Section 2.1 is on foundation and formulations of refinement. Section 2.2 is on data refinement. Section 2.3 is on a recent development in the area, refinement of classes in object-oriented programming. Section 2.4 lists some monographs and textbooks that collect most of the works as organized, complete expositions.

2.1 Foundation and Program Refinement

1. Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

   Describes a method of formalizing program semantics and proving correctness of programs. A structured imperative program is represented by a flowchart, with each edge annotated with a proposition about program variables when execution reaches that point. Thus, each program statement S has some propositions P at its entrance and some propositions Q at its exit. Proving the program correct is then verifying for each S that if P holds upon entrance of S, then Q holds upon exit of S according to some given axioms. In the presence of loops, termination is proved by a decreasing variant.

2. (a) Christopher Strachey. Towards a formal semantics. In T. B. Steel, Jr., editor, *Formal Language Description Languages for Computer Programming*, pages 198–220. North-Holland, 1966.

   (b) Dana Scott. Outline of a mathematical theory of computation. In *Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems*, pages 169–176. Princeton University, 1970. Also as Technical Monograph PRG-2, Oxford University Computing Laboratory, Programming Research Group.

   (c) Dana Scott. The lattice of flow diagrams. In Erwin Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 311–366. Springer, 1971. Also as Technical Monograph PRG-3, Oxford University Computing Laboratory, Programming Research Group.

   (d) Dana Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. In Jerome Fox, editor, *Proceedings of the Symposium on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*, pages 19–46. Polytechnic Press of the Polytechnic Institute of Brooklyn, 1971.

   These works propose defining programming language semantics by mapping each piece of program syntax to a mathematical object. A command could be mapped to a function from states to states, an expression with variables and possible side effects could be mapped to a function from states to numbers and states, etc. The mapping is defined structurally over the grammar of the programming language. Recursive programs are treated by imposing a lattice structure over the mathematical objects and using fixpoints. If the programming language allows commands or functions to be passed around as values, the mathematical objects need to be restricted further (namely, continuous functions in the lattice theoretic sense) to avoid a Russell paradox or cardinality problem.

3. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

   Postulates elementary axiom schemas for proving programs correct in the same vein as [1], but on textual programs instead of flowcharts. They are given as triples of the form $p\{S\}q$, meaning that if precondition p holds to begin with, then program S establishes postcondition q. Some examples:

$$(q \text{ with } v \text{ replaced by } E)\{v{:=}\,E\}q$$
$$(q \wedge c)\{S\}q \Rightarrow q\{\textbf{while } c \textbf{ do } S\}(\neg c \wedge q)$$

In this treatment, conditions for termination and successful delivery of results are omitted because program execution could as well be aborted due to arbitrary limits such as arithmetic overflow and memory exhaustion, but it is noted that the axioms can be adapted so that they are restricted to successful and terminating executions.

Although the theory in this paper is already given in [1], this paper adds a discussion of the benefits of basing both program semantics and program correctness proofs on axiomatic approaches similar to the above.

4. Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

Extends the method in [3] by defining, for each program S and postcondition q, the *weakest* (necessary) precondition p such that $p\{S\}q$ holds and S terminates. (We assume that the only cause of nontermination is infinite loops.) This is then known as the weakest precondition semantics of programs. Some examples (denoting such weakest preconditions by $\mathrm{wp}\, S\, q$):

$$\begin{aligned}\mathrm{wp}\,(v{:=}\,E)\,q &= \text{replace } v \text{ by } E \text{ in } q\\ \mathrm{wp}\,(\textbf{while } c \textbf{ do } S)\,q &= \exists k \geq 0 \cdot h_k(q)\end{aligned}$$

where

$$\begin{aligned}h_0(q) &= \neg c \wedge q\\ h_{k+1}(q) &= h_0(q) \vee (c \wedge \mathrm{wp}\, S\, h_k(q))\\ &= (\neg c \wedge q) \vee (c \wedge \mathrm{wp}\, S\, h_k(q))\end{aligned}$$

We will see weakest preconditions for common program constructs in section 4.2.

Because computing the weakest precondition of a loop is usually difficult and unnecessary, sufficiency theorems for loop correctness, namely, the use of loop invariants and variants, are also stated.

Remark: Dijkstra has a more general do-loop than the while-loop above; the do-loop accommodates an arbitrary number of pairs of condition c_i and body S_i rather than just one condition c and one body S.

5. R. J. R. Back. *On the Correctness of Refinement Steps in Program Development.* PhD thesis, Department of Computer Science, University of Helsinki, 1978.

Defines *refinement* formally for programs with state transformer semantics and characterizes it in terms of weakest preconditions. Illustrates how it justifies top-down programming and program transformations with refinement theorems and examples.

Using Γ to stand for the state space, a program is a function from Γ to $\mathcal{P}(\Gamma) - \{\emptyset\}$, i.e., it maps a state to a non-empty set of states; a special state $\infty \in \Gamma$ stands for the state of nontermination, and a program is required to map ∞ to $\{\infty\}$. (Back's definition is actually more general: a program may be a function from Γ to $\mathcal{P}(\Delta) - \{\emptyset\}$, where Δ is another state space.) Given a program P and a predicate $q \colon \Gamma \to \mathrm{Bool}$ with $q(\infty) = \bot$, the weakest precondition $\mathrm{wp}\, P\, q$ is the predicate defined by $\lambda s \in \Gamma \cdot \forall s' \in P(s) \cdot q(s')$.

Program P is refined by program P' iff for any state $s \in \Gamma$, $\infty \notin P(s)$ implies $P'(s) \subseteq P(s)$. We can then prove that this is equivalent to: for any $q \colon \Gamma \to \mathrm{Bool}$ with $q(\infty) = \bot$, $\mathrm{wp}\, P\, q \Rightarrow \mathrm{wp}\, P'\, q$.

A specification can also be seen as a state transformer like a program is, specifying the acceptable set of final states for each initial state. This is the first time a unification of specification and program is hinted.

6. R. J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, August 1981.

   Defines *refinement* formally in a general and abstract setting, explores its consequences and properties, and characterizes the extra conditions under which it supports specification statements (for top-down programming), program transformations, and various other desirable refinement laws.

   The general and abstract definition of refinement: given a space of programs, a space of specifications *Spec*, and a satisfaction relation between them, we say P is refined by P' iff for all specification $S \in \mathit{Spec}$, P satisfies S implies P' satisfies S.

   The paper also shows what this definition becomes when instantiated to various specific program semantics, e.g., state transformers, predicate transformers, etc.

7. Eric C. R. Hehner. **do** considered **od**: A contribution to the programming calculus. *Acta Informatica*, 11(3):287–304, March 1979.

   Defines the weakest precondition semantics of recursion by strongest fixed point, and suggests the use of recursion in place of loops in programming in order to eliminate dilemmas in practical uses of structured loops due to the need for early exits.

   For a recursive program named P and defined as $F(P)$, i.e., as a compound statement that invokes P recursively, the semantics is defined by the strongest solution to $\mathrm{wp}\, P\, q$ of the equation

   $$\mathrm{wp}\, P\, q = \mathrm{wp}\, F(P)\, q$$

This is a fixed point equation, since the right hand side expands to an expression involving $\mathrm{wp}\, P\, q$.

8. Patrick Cousot and Radhia Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.

   Demonstrates that given a monotonic function f over a complete lattice L, fixed points of f can be computed by iterative construction over transfinite ordinals. For example, starting with the bottom of L, define a sequence s by:

   $$\begin{aligned} s(0) &= \bot \\ s(n+1) &= f(s(n)) && \text{for ordinal } n \\ s(\alpha) &= \bigsqcup_{\beta<\alpha} s(\beta) && \text{for limit ordinal } \alpha \end{aligned}$$

   then the least fixed point of f will be $s(\gamma)$ for some ordinal γ.

   Transfinite ordinals are needed because f is not known to be continuous; if f is continuous, then we only need to iterate up to $s(\omega)$.

   This mathematical result is relevant in the following way. Program constructs listed in [4] support bounded nondeterminism but not unbounded nondeterminism, i.e., although a program may make nondeterministic choices, each time the number of options is finite. In a subsequent writing, Dijkstra shows that weakest precondition predicate transformers of program constructs (which are designed to be monotonic functions for support of refinement) are continuous functions precisely when they lack unbounded nondeterminism. This is why the semantics of the loop in [4] needs only be an existential quantifier over the natural numbers, i.e., just $s(\omega)$. As we will see below, program constructs supporting unbounded nondeterminism will be introduced for various reasons, and the semantics of the loop will require the use of transfinite ordinals.

9. H. J. Boom. A weaker precondition for loops. *ACM Transactions on Programming Languages and Systems*, 4(4):668–677, October 1982.

   Argues that unbounded nondeterminism is a useful abstraction, and redefines the weakest precondition semantics of the loop accordingly:

   $$\mathrm{wp}\,(\textbf{while}\ c\ \textbf{do}\ S)\, q = \exists \alpha \cdot h_\alpha(q)$$

   where

   $$h_\alpha(q) = (\neg c \wedge q) \vee (\exists \beta < \alpha \cdot c \wedge \mathrm{wp}\, S\, h_\beta(q))$$

   (If we choose $\alpha = \omega$, this definition reduces to the original one in [4].) The paper then proves the correctness of this definition via the operational semantics of a certain nondeterministic machine, and shows some applications: the termination of a loop-based implementation of Ackermann's function, and the definition of fair loop command that interleaves processes without starvation.

Remark: Like [4], this paper gives the semantics of the general do-loop; we have simplified it to the while-loop here.

10. Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

    Defines *refinement* via the weakest precondition predicate transformer, which coincides with the characterization in [5]. Also defines the weakest precondition semantics of the specification statement:

    $$\mathrm{wp}\,[pre, post]\,q = pre \wedge (\forall v \cdot post \Rightarrow q)$$

    where v stands for all program variables; and of recursion: if P is a program recursively defined as $F(P)$, then it is the least fixed point of F:

    $$P = \mu X.F(X)$$

    where "least" means least refined in the refinement ordering and "=" means both sides refine each other. If we consider the weakest preconditions of both sides of the equation, we get the same definition as in [7].

    The specification statement allows unbounded nondeterminism. As a consequence, properties and refinement laws of recursion involve transfinite ordinals, such as computing the least fixed point by iteration as in [9].

    The specification statement unifies programs and specifications by extending the programming language.

11. Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

    Defines *refinement* via the weakest precondition predicate transformer, which coincides with the characterization in [5] and the definition in [10]. Also defines the weakest precondition semantics of several forms of specification statements. The simplest form is the same as in [10], and the most general form, $v\colon [pre, post]$, specifies which variables may be changed and allows referring to initial values.

    We will see details and examples in section 4.2.

    Again, the specification statement unifies programs and specifications by extending the programming language.

12. E. C. R. Hehner and C. A. R. Hoare. A more complete model of communicating processes. *Theoretical Computer Science*, 26(1,2):105–120, September 1983.

    Previously, semantics of Hoare's communicating sequential processes was defined by axioms of these forms:

    $$P \text{ sat } R = \text{some transformation of } R$$
    $$P \text{ sat } R \Leftarrow \text{some transformation of } R$$

where P is the process being defined and R is any specification in terms of communication history and the set of possible next messages. These axioms state the sufficient and perhaps also necessary conditions under which R is true of the past and the current state of P. This is similar in spirit to [3,4].

This paper identifies a process P with the strongest specification R such that P sat R holds. Thus, a process is a predicate in the relevant variables, which are the history and the set of next messages in this case. Then satisfaction of a specification, which is again a predicate in the same variables, is implication.

In this way, processes and specifications are unified by embedding processes into the specification language (the language of predicate logic).

13. Eric C. R. Hehner. Predicative programming, part I. *Communications of the ACM*, 27(2):134–143, February 1984.

    A program is defined as a predicate in the relevant variables: initial and final values of state variables. The actual definitions are different from the modern ones: the semantic definitions of assignment statements, sequential compositions, and if-statements all bear healthiness constraints such as definedness (e.g., x/y is defined iff $y \neq 0$) and termination. Refinement is implication with side conditions, and associativity of sequential composition holds conditionally. In an appendix, Hoare's use of a "stop" variable standing for termination is introduced, and unconditional associativity is restored with some other loss.

    Again, this unifies programs and specifications by embedding programs into the specification language (the language of predicate logic).

14. Eric C. R. Hehner. Termination is timing. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 36–47, Groningen, The Netherlands, June 1989. Springer.

    Eric C. R. Hehner. A practical theory of programming. *Science of Computer Programming*, 14(2,3):133–158, October 1990.

    Simplifies the formulation in [13] by omitting definedness and separating out the requirement of termination as an additional specification on time bound. As a result, the definitions and the calculus are straightforward, sequential composition is associative (it is equivalent to composition of relations), and refinement is simply implication.

    We will see details and examples in section 4.1.

15. Leslie Lamport. Temporal logic of actions. Research Report 79, Systems Research Center, Digital Equipment Corporation, December 1991. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

Like [14], uses predicates of initial and final values of state variables as specifications and semantics of programs, and adds temporal and frame operators to describe reactive systems. Different from [14], however, explicit time bounds are omitted because temporal operators are available.

16. Eric C. R. Hehner. Formalization of time and space. *Formal Aspects of Computing*, 10:290–306, 1998.

    Shows how to account for memory usage (including both maximum and average) in the same way [14] accounts for time. Thus, a program specification can specify result, time usage, and space usage in the same refinement calculus.

17. Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall International Series in Computer Science. Prentice Hall, 1997.

    A theory of functional programming. Specifications and programs are functions and relations. Derivation of programs from specifications is usually equational reasoning rather than refinement, and the category theory of functions and relations is used for such derivations.

18. Alexander Bunkenburg. *Expression Refinement*. PhD Thesis, Department of Computing Science, University of Glasgow, 1997.

    Develops a refinement calculus for functional programming in which a notion of refinement between expressions (rather than commands) is defined. A four-valued equational logic, rather than classical logic, is used.

19. Douglas R. Smith. Mechanizing the development of software. In Manfred Broy, editor, *Calculational System Design*, volume 173 of *NATO ASI Series F: Computer and Systems Sciences*. IOS Press, 1999.

    A specification is an algebraic theory of constructors, operations, and axioms. A category theory of algebras is used to derive programs: refinement is morphism.

20. C. A. R. Hoare and He Jifeng. Unifying theories for logic programming. In Tony Hoare, Manfred Broy, and Ralf Steinbrüggen, editors, *Engineering Theories of Software Construction*, volume 180 of *NATO Science Series III: Computer and Systems Sciences*, pages 21–45. IOS Press, 2001.

    Defines several predicate semantics (from a simple but weak one to a complicated but strong one) for logic programming, leading to algebraic laws suitable for a refinement calculus.

21. Ingrid Rewitzky and Chris Brink. Unification of four versions of program semantics. *Formal Aspects of Computing*, 10(1):1–29, 1998.

    Equivalence of the relation view, the predicate transformer view, the information system view, and the Smyth powerdomain view, using fairly recent results in lattice theory.

22. J. M. Morris and A. Bunkenburg. Term transformer semantics. Submitted to*ACM Transactions on Programming Languages and Systems*, 1999.

Generalizes the predicate transformer semantics (a command is a function that transforms a boolean-valued function to a boolean-valued function) to a term transformer semantics: the semantics of an expression is a function that transforms a T-valued function to a T-valued function, where T is an appropriate type. This generalization is useful for unifying imperative programming and functional programming in both semantics and refinement.

2.2 Data Refinement

23. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

24. C. A. R. Hoare, J. F. He, and J. W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71–76, May 1987.

Defines two special cases of data refinement—downward simulation by a relation from abstract variables to concrete variables and, dually, upward simulation by a relation from concrete variables to abstract variables—and proves that their combination covers all data refinements for programs in which nondeterminism is bounded.

25. C. C. Morgan. Auxiliary variables in data refinement. *Information Processing Letters*, 29(6):293–296, December 1988.

Describes and justifies a procedure of data refinement by first introducing concrete variables, then refining each statement to remove data dependency on abstract variables, and finally eliminating abstract variables. The refinements necessary are carried out using an invariant that relates concrete variables with abstract variables.

The paper is thus named because concrete variables are auxiliary variables at the beginning, and abstract variables become auxiliary variables at the end.

26. J. M. Morris. Laws of data refinement. *Acta Informatica*, 26(4):287–308, 1989.

Defines a sufficient condition for data refinement between programs based on an invariant that relations concrete variables with abstract variables. Furthermore, proves a collection of laws for data-refining a program structurally; using these laws, one needs only calculate with predicate logic.

27. Wei Chen and Jan Tijmen Udding. Towards a calculus of data refinement. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 197–218, Groningen, The Netherlands, June 1989. Springer.

Proves the equivalence of several conditions for data refinement, one of which is formulated by the authors and the rest formulated previously by others. Defines a condition for data refinement in context. States a collection of laws for data-refining a program structurally in context.

28. C. C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.

    Describes and justifies a procedure of data refinement by going through the structure of a program and calculationally transforming the relevant parts, given an invariant that relates concrete variables with abstract variables. This is similar to the laws in [26].

    Also gives a weakest precondition predicate transformer semantics of logical constant introduction, which is a dual of local variable introduction. Some data refinements in this paper are performed with logical constant introduction and therefore different from those in [26].

29. P. H. B. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.

    Defines a method of data refinement by a predicate transformer relating the abstract state space with the concrete state space: given a postcondition about the abstract variables, the transformer maps it to a precondition about the concrete variables.

30. P. H. B. Gardiner and Carroll Morgan. A single complete rule for data refinement. *Formal Aspects of Computing*, 5(4):367–382, 1993.

    Points out that the definition in [29] corresponds to upward simulation in [24] and shows that all correctness-preserving data refinements are expressible this way, provided nondeterminism in the abstract operations is bounded.

2.3 Class/Object Refinement

31. Richard F. Paige and Jonathan S. Ostroff. An object-oriented refinement calculus. Technical Report CS-1999-07, Department of Computer Science, York University, Canada, December 1999. Submitted to *Formal Aspects of Computing*.

    Defines refinement for object-oriented programming in the Eiffel sense using a predicative semantics similar to [14] for ease of embedding into automated theorem provers, but adopting the syntax of the specification statement in [11] for supporting the well-known idea in object-oriented programming of *design by contract*, which specifies every method by a precondition and a postcondition. The specification statement is therefore given a predicative semantics:

    $$v\colon [P, Q] \;=\; P \Rightarrow Q \wedge x' = x \wedge t' \geq t$$

where x stands for state variables not mentioned in v.

Each method comes with a precondition P and a postcondition Q. An implementation of a method is simply required to refine $t, a\colon [P, Q]$, where t is the time variable and a stands for the attributes of the object; if it invokes other methods, it may assume their correctness. An invocation of a method expands to a specification statement derived from P and Q (the details are lengthy because a method invocation whose parameters come from method invocations needs to be carefully handled). Open recursion does not pose a termination problem because one can in principle specify and verify time bounds like in [14].

This work does not address the representation exposure problem: the problem of writing a class specification without mentioning its internal state variables. This work, in fact, exposes all internal attributes.

32. Richard F. Paige and Eric C. R. Hehner. Bunches for object-oriented, concurrent, and real-time specification. In *FM'99 - Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 530–550, Toulouse, France, September 1999. Springer.

    Embeds object-oriented programming into the framework in [14,37] so as to take advantage of the available theories of concurrency and timing in reasoning about object-oriented programs.

    A class is defined as a bunch[1] of records of the same type (providing the same attributes and methods) satisfying a class invariant, with methods specified as possibly nondeterministic functions. This definition works best when objects are immutable and methods are purely functional, so the usual method of mimicking state updates by functional updates is again used, i.e., the imperative method invocation `o.m()` in programming languages is written as the assignment statement $o{:=}\,o.m$ in the theory. Inheriting a class subbunches the parent class, which permits strengthening existing methods, adding extra methods and attributes, and strengthening class invariants.

    This work does not directly address the representation exposure problem; in fact it exposes internal attributes. However, one can argue that given the framework of [37], data refinement can be carried out, and therefore the attributes given in a class specification can hide the actual state variables by data abstraction.

33. Ralph-Johan Back, Anna Mikhajlova, and Joakim von Wright. Class refinement as semantics of correct object substitutability. *Formal Aspects of Computing*, 12:18–40, 2000. Also as TUCS Technical Report 333.

    Extends the refinement calculus of [36] to object classes, so as to characterize the conditions under which a subclass refines the behaviours of its superclass and therefore is a valid substitute from the client's point

[1] A bunch is essentially a set without enclosing braces.

of view. In short, the subclass is required to be a data refinement of the superclass.

A class C is defined by its attributes, constructor K, and methods M_i. The constructor and the methods are procedural specifications.

A subclass C' may add and replace attributes, provide its own constructor K', and override methods M_i by M_i'. It refines C and is proved to be a valid substitute if:

- the signature of M_i' is a subtype of the signature of M_i, i.e., contravariant in parameters and covariant in result
- K' data-refines K and M_i' data-refines M_i using the same data relation R between the attributes of C' and those of C

The subclass C' may also introduce new methods N_j. If N_j does not preserve some class invariant implied by the constructor and method specifications of C, then C' is not a valid substitute. Thus class refinement requires further that:

- N_j preserves every predicate established by K and preserved by M_i (under data refinement by the same R above)

Since this condition is difficult to use, a sufficient condition is also stated and proved:

- N_j refines a certainly demonically chosen sequence of statements composed of calls to M_i or M_i' and other statements that do not modify the attributes of C' (under data refinement by R)

2.4 Monographs

34. Carroll Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.

    Contains a large collection of refinement laws for practical use and several case studies of developments of medium-scale programs. Its treatment of data refinement follows that in [25].

35. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1998.

    Uses a unified framework to explain and compare all known formulations and methods of data refinement.

36. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer, 1998.

Explains refinement calculus as started in [5] and further developed since then, using such mathematical machinery as lattice theory and high-order logic. Since lattice theory suggests generalizations by symmetry and duality, it follows that many familiar program constructs have unfamiliar duals; therefore, this exposition treats them as well and in fact treats the theory of refinement calculus and program semantics in as much generality as permitted by lattice theory, without bias to any artificially chosen set of program constructs.

37. Eric C. R. Hehner. *A Practical Theory of Programming.* Texts and Monographs in Computer Science. Springer, 1993.

    A full exposition of the formulation in [14], with additions including definitions and construction methods of fixed points (for both recursive programs and recursively defined types), data type abstraction and program abstraction, data refinement, concurrency, and communication (input/output). The condition for data refinement in this book is equivalent to those in [27].

38. Chris Brink and Ingrid Rewitsky. *A Paradigm for Program Semantics: Power Structures and Duality.* The University of Chicago Press, 2001.

    Develops in detail and in the context of lattice theory and domain theory four programming semantics—relations, predicate transformers, information systems, and domain theory—and shows how they can be translated to each other.

3 Notations

We will manipulate sums and maxima over ranges and list segments abundantly. To streamline the calculations, we need an optimized, consistent, and yet versatile notation. The notation used in this paper is shown in the "Optimized" column in the following table, with their meaning and conventional equivalence are shown in the other two columns:

Meaning	Conventional	Optimized
Sum of segment	$\sum_{i=0}^{9} L(i)$	$\Sigma L[0;..10]$
Maximum	$\max(x, \max(y, z))$	$x \uparrow y \uparrow z$
Maximization	$\max_{i=0}^{9} i+2$	$\max\langle i\colon 0,..10 \to i+2\rangle$
Anonymous function	$\lambda v \cdot v+2$	$\langle v \to v+2\rangle$
Substitution	$(v+2)_v^{42} = 42{+}2$	$\langle v \to v+2\rangle(42) = 42+2$

Our notation for maximization is longer because we optimize it to clarify its scope and to show its bounds in a normal font. Our notations for range ($0,..10$ and $0;..10$) are inclusive on the lower bound and exclusive on the upper bound because the length of the range is apparent (just subtract) and the empty range is not written as something odd like $0\ldots-1$. Our notation for anonymous function (lambda abstraction) revives Church's original intention to mark the scope of the local variable. Given a notation for anonymous function, substitution can be unambiguously expressed as function construction and application without introducing extra notation or convention.

4 Formal Refinements

This section describes and illustrates two refinement calculi, using one common example throughout. The common example is the maximum segment sum problem: Given an array or list L of integers, find the maximum over sums of its segments. In notation, the maximum segment sum of L is

$$\max\langle i\colon 0,..\#L+1 \to \max\langle j\colon i,..\#L+1 \to \Sigma L[i;..j]\rangle\rangle$$

If we abbreviate the inner maximization as *inner* and the outer one as *outer*:

$$(1) \qquad inner(x,y) = \max\langle j\colon x,..y+1 \to \Sigma L[x;..j]\rangle$$
$$(2) \qquad outer(z) = \max\langle i\colon 0,..z+1 \to inner(i,z)\rangle$$

then the maximum segment sum can be rewritten as

$$outer(\#L)$$

In order to discover a fast iterative solution, we try to use $outer(k)$ to help us compute $outer(k+1)$. If we have stored the former, we hope to incrementally compute the latter. The following calculation discovers recurrence equations for this incremental computation, and we perform it now because it is invariant under change of formal methods.

We first calculate $inner(i,k+1)$:

$$\begin{array}{ll}
\quad inner(i,k+1) & \text{definition} \\
= \max\langle j\colon i,..k+2 \to \Sigma L[i;..j]\rangle & \text{unfold once} \\
= \max\langle j\colon i,..k+1 \to \Sigma L[i;..j]\rangle \uparrow \Sigma L[i;..k+1] & \text{definition} \\
= inner(i,k) \uparrow \Sigma L[i;..k+1] &
\end{array}$$

Then we calculate $outer(k+1)$:

$$\begin{array}{ll}
\quad outer(k+1) & \text{definition} \\
= \max\langle i\colon 0,..k+2 \to inner(i,k+1)\rangle & \text{from above} \\
= \max\langle i\colon 0,..k+2 \to inner(i,k) \uparrow \Sigma L[i;..k+1]\rangle & \text{distribute} \\
= \quad \max\langle i\colon 0,..k+2 \to inner(i,k)\rangle & \text{unfold once} \\
\quad \uparrow \max\langle i\colon 0,..k+2 \to \Sigma L[i;..k+1]\rangle & \\
= \quad \max\langle i\colon 0,..k+1 \to inner(i,k)\rangle & \text{empty max} \\
\quad \uparrow inner(k+1,k) & \\
\quad \uparrow \max\langle i\colon 0,..k+2 \to \Sigma L[i;..k+1]\rangle & \\
= \quad \max\langle i\colon 0,..k+1 \to inner(i,k)\rangle \uparrow -\infty & \text{identity element} \\
\quad \uparrow \max\langle i\colon 0,..k+2 \to \Sigma L[i;..k+1]\rangle & \\
= \quad \max\langle i\colon 0,..k+1 \to inner(i,k)\rangle & \text{definition} \\
\quad \uparrow \max\langle i\colon 0,..k+2 \to \Sigma L[i;..k+1]\rangle & \\
= outer(k) \uparrow \max\langle i\colon 0,..k+2 \to \Sigma L[i;..k+1]\rangle &
\end{array}$$

This formula enables the desired incremental computation, except that it involves an expensive maximization. Can we replace it by another incremental computation? Yes. Define:

(3) $$extra(z) = \max\langle i\colon 0,..z+1 \to \Sigma L[i;..z]\rangle$$

Then we have:

$$\begin{array}{ll}
\quad extra(k+1) & \text{definition} \\
= \max\langle i\colon 0,..k+2 \to \Sigma L[i;..k+1]\rangle & \text{unfold once} \\
= \max\langle i\colon 0,..k+1 \to \Sigma L[i;..k+1]\rangle \uparrow \Sigma L[k+1;..k+1] & \text{empty sum} \\
= \max\langle i\colon 0,..k+1 \to \Sigma L[i;..k+1]\rangle \uparrow 0 & \text{unfold, factor} \\
= \big(\max\langle i\colon 0,..k+1 \to \Sigma L[i;..k]\rangle + L(k)\big) \uparrow 0 & \text{definition} \\
= \big(extra(k) + L(k)\big) \uparrow 0 &
\end{array}$$

Thus we have the following recurrence equations:

(4) $$outer(k+1) = outer(k) \uparrow extra(k+1)$$

(5) $$extra(k+1) = \big(extra(k) + L(k)\big) \uparrow 0$$

and their base cases:

(6) $$outer(0) = 0$$

(7) $$extra(0) = 0$$

Intuitively, $outer(k)$ is the solution to the subproblem where the length of the given list is k instead of $\#L$, and $extra(k)$ is the maximum over the sums of all segments that end at index k.

In the following program developments, we use variables k to step through the range $0,..\#L$, s to hold $outer(k)$ (and therefore the final answer at the end), and c to hold $extra(k)$.

4.1 Predicate Semantics

In this section, we describe the refinement calculus formulated in [14].

A specification is a boolean expression relating initial values of program variables with their final values.[2] Initial values are denoted by the names of the variables themselves, and final values by adding primes to names. The specification for maximum segment sum is written as:

$$s' = outer(\#L)$$

A program is a special case of a specification, since a program specifies the relation between initial values and final values, albeit algorithmically. Here are some program constructs and their respective boolean expressions:

- **ok** does nothing. It stands for

  $$s' = s \wedge c' = c \wedge k' = k$$

- $s{:=}\,E$ assigns the value of an expression E to the variable s. It stands for

  $$s' = E \wedge c' = c \wedge k' = k$$

  If E is readily computed by the computer, then $s{:=}\,E$ is a program; otherwise, it is still a specification defined by the above.

- $P \,.\, Q$ executes P and then Q. It stands for

  $$\exists s'', c'', k'' \cdot \langle s', c', k' \rightarrow P\rangle(s'', c'', k'') \wedge \langle s, c, k \rightarrow Q\rangle(s'', c'', k'')$$

  If P and Q are programs, then $P \,.\, Q$ is a program; otherwise, it is still a specification defined by the above.

- **if** c **then** P **else** Q executes P if c is true, and Q otherwise. It simply stands for

  $$(c \wedge P) \vee (\neg c \wedge Q)$$

  If P and Q are programs, then **if** c **then** P **else** Q is a program; otherwise, it is still a specification defined by the above.

Loop constructs are missing because we will use recursion; the example refinement in the following will illustrate how.

A useful theorem is the *substitution law*:

(8) $$s{:=}\,E \,.\, P \;=\; \langle s \rightarrow P\rangle(E)$$

Refinement is defined as implication: P is refined by Q iff $P \Leftarrow Q$. Stepwise refinement is supported by this definition because implication is a partial order,

[2] In general, a specification is a boolean expression in terms of all observables we are interested in. In this paper, we are just interested in initial and final values of state variables and time. Some other observables are space and communication channels.

and the program constructs are monotonic with respect to it. For example, if we have $A \Leftarrow A_1$ and $B \Leftarrow B_1$, then we can conclude $A\ .\ B \Leftarrow A_1\ .\ B_1$.

The solution to the maximum segment sum goes as follows. Define:

$$\begin{aligned} \mathit{MSS} &= s' = \mathit{outer}(\#L) \\ M &= s = \mathit{outer}(k) \wedge c = \mathit{extra}(k) \wedge 0 \le k \le \#L \end{aligned}$$

Then we refine:

$$\begin{array}{lll} & & \mathit{MSS} \\ (9) & \Longleftarrow & s{:=}\,0\ .\ c{:=}\,0\ .\ k{:=}\,0\ .\ M \Rightarrow \mathit{MSS} \end{array}$$

$$\begin{array}{lll} & & M \Rightarrow \mathit{MSS} \\ (10) & \Longleftarrow & \textbf{if } k = \#L \textbf{ then ok} \\ & & \textbf{else } (c{:=}\,(c + L(k)) \uparrow 0\ .\ s{:=}\,s \uparrow c\ .\ k{:=}\,k+1\ .\ M \Rightarrow \mathit{MSS}) \end{array}$$

In refinement (10), we implement $M \Rightarrow \mathit{MSS}$ by a program that invokes itself recursively. This is how we achieve iteration by recursion. That it terminates and delivers results will be treated after we prove the above two refinements.

The refinements to prove are (9) and (10). The former is proved by the substitution law (8) and the base equations (6) and (7). The latter can be proved by the following two refinements:

$$\begin{array}{ll} (11) & M \Rightarrow \mathit{MSS} \Longleftarrow k = \#L \wedge \textbf{ok} \\ (12) & M \Rightarrow \mathit{MSS} \Longleftarrow k \ne \#L \wedge (c{:=}\,(c + L(k)) \uparrow 0\ .\ s{:=}\,s \uparrow c\ . \\ & \qquad\qquad\qquad\qquad k{:=}\,k+1\ .\ M \Rightarrow \mathit{MSS}) \end{array}$$

Refinement (11) is proved by expanding **ok**. Refinement (12) is the heart of the solution. By the substitution law (8), it is equivalent to:

$$M \Rightarrow \mathit{MSS} \Longleftarrow k \ne \#L \wedge (M_2 \Rightarrow \mathit{MSS})$$

where M_2 is M after substitutions:

$$\begin{aligned} M_2 &= \langle s, c, k \to M \rangle \big(s \uparrow (c + L(k)) \uparrow 0, (c + L(k)) \uparrow 0, k+1\big) \\ &= \quad s \uparrow (c + L(k)) \uparrow 0 = \mathit{outer}(k+1) \\ &\quad \wedge\ (c + L(k)) \uparrow 0 = \mathit{extra}(k+1) \\ &\quad \wedge\ 0 \le k+1 \le \#L \end{aligned}$$

To finish the proof, it suffices to prove the following:

$$k \ne \#L \wedge M \implies M_2$$

i.e.,

$$(13) \qquad \begin{pmatrix} \quad s = \mathit{outer}(k) \\ \wedge\, c = \mathit{extra}(k) \\ \wedge\, 0 \le k < \#L \end{pmatrix} \implies \begin{pmatrix} \quad s \uparrow (c + L(k)) \uparrow 0 = \mathit{outer}(k+1) \\ \wedge\, (c + L(k)) \uparrow 0 = \mathit{extra}(k+1) \\ \wedge\, 0 \le k+1 \le \#L \end{pmatrix}$$

which is an application of simple arithmetic and equations (4) and (5).

The above development does not address termination. Without requiring termination, and with recursion allowed in a program, one could refine any specification P by $P \Leftarrow P$ and declare to have finished the programming. To address termination, we add a time bound requirement to the specification (all well-written specifications come with time bounds) and prove that it is met in refinements. This can be included in the theory in the following way:

1. Introduce a program variable t in addition to s, c, and k. The type of t could be any numeric type with or without infinities depending on whether we want to model infinite computations or not. Its intention could be either a real time clock or a cost counter.

2. Write time bound specifications in the form $t' - t \leq \mathit{bound}$. For example, if we wish to solve the maximum segment sum problem in time linear to $\#L$, we may specify
$$\mathit{MSS} \wedge t' - t \leq \#L$$
It is implicit that $t' - t \geq 0$ must also hold.

3. Insert $t{:=}\, t + 1$ or similar expressions at places in the program that are considered to consume time, e.g., before or after significant operations, or before recursion.

4. With the addition of the program variable t, the definitions of some programming constructs change accordingly:
$$\begin{aligned}
\textbf{ok} &= s' = s \wedge c' = c \wedge k' = k \wedge t' = t \\
s{:=}\, E &= s' = E \wedge c' = c \wedge k' = k \wedge t' = t \\
P \,.\, Q &= \exists s'', c'', k'', t'' \cdot \quad \langle s', c', k', t' \rightarrow P\rangle(s'', c'', k'', t'') \\
&\qquad\qquad\qquad\quad \wedge \langle s, c, k, t \rightarrow Q\rangle(s'', c'', k'', t'')
\end{aligned}$$

Note that we do not reset t to 0 at the beginning of the program and specify $t' \leq \mathit{bound}$ for its running time. If we view t as a real time clock, we are simply forbidden from resetting it; and if we view it as a counter, it is still important to refrain from resetting it because our program may be part of a larger refinement with a global time bound. We therefore specify the time difference $t' - t$ instead.

Here are the specifications and refinements including time; we count the number of recursions:

$$\begin{aligned}
&\mathit{MSS} \wedge t' - t \leq \#L \\
\Leftarrow\ & s{:=}\, 0 \,.\, c{:=}\, 0 \,.\, k{:=}\, 0 \,.\, M \Rightarrow \mathit{MSS} \wedge t' - t \leq \#L - k
\end{aligned}$$

$$\begin{aligned}
&M \Rightarrow \mathit{MSS} \wedge t' - t \leq \#L - k \\
\Leftarrow\ & \textbf{if } k = \#L \textbf{ then ok} \\
& \textbf{else } (c{:=}\, (c + L[k]) \uparrow 0 \,.\, s{:=}\, s \uparrow c \,.\, k{:=}\, k + 1 \,. \\
& \qquad t{:=}\, t + 1 \,.\, M \Rightarrow \mathit{MSS} \wedge t' - t \leq \#L - k)
\end{aligned}$$

We do not need to prove them from scratch. By monotonicity of program constructions as well as $\wedge$, we can split them into two sets of refinements: one set is just like (9) and (10) and is already proved, and the other set is the following:

$$\begin{array}{ll} & t' - t \leq \#L \\ \Leftarrow & s{:=}\,0 \ . \ c{:=}\,0 \ . \ k{:=}\,0 \ . \ 0 \leq k \leq \#L \Rightarrow t' - t \leq \#L - k \end{array}$$

$$\begin{array}{ll} & 0 \leq k \leq \#L \Rightarrow t' - t \leq \#L - k \\ \Leftarrow & \textbf{if } k = \#L \textbf{ then ok} \\ & \textbf{else } (c{:=}\,(c + L[k]) \uparrow 0 \ . \ s{:=}\,s \uparrow c \ . \ k{:=}\,k+1 \ . \\ & \qquad t{:=}\,t+1 \ . \ 0 \leq k \leq \#L \Rightarrow t' - t \leq \#L - k) \end{array}$$

and their proofs are straightforward.

4.2 Predicate Transformer Semantics

In this section, we describe refinement originally formulated in [5], using modern notations and definitions from [10,11].

Specification and program are again unified, but here the programming language is extended to provide a notation for specification. Here are some program constructs and their informal meaning (their formal semantics will be given next):

- **ok** does nothing.
- $v{:=}\,E$ assigns the value of E to the state variable v.

  Sometimes v stands for a list of state variables and E stands for a corresponding list of expressions, in which case we mean parallel assignment. Sometimes we write out the lists partially or completely, e.g., $k, s, c{:=}\,0, 1, 2$ and $k, v{:=}\,0, E$.
- $P \ . \ Q$ executes P and then Q.
- **if** c **then** P **else** Q is the conditional statement.
- **while** c **do** P is the while-loop.
- $v \colon [\mathit{pre}, \mathit{post}]$ is the specification statement: *pre* is a precondition, and *post* is a postcondition; they are boolean expressions in the state variables. If *pre* is true to begin with, then *post* will be established at the end without modifying any state variable other than those mentioned in the list v. If *pre* is false to begin with, then any behaviour is allowed, including nontermination.

  If we wish to mention both initial and final values of state variables in the postcondition, for example to specify that s is to be decremented by 1 if s was at least 1, we may write s_0 for the initial value of s in *post* thusly: $s \colon [s \geq 1, s = s_0 - 1]$.

The formal semantics of the program constructs are given via weakest preconditions: given a statement P and a boolean expression u, we can ask for the weakest boolean expression w such that P establishes postcondition u under precondition w. We can then say that P represents a function that maps u to w and name this function $\mathrm{wp}\,P$ or simply P. Here are the weakest precondition semantics of the program constructs:

$$\begin{aligned}
\mathrm{wp}\,\mathbf{ok}\,u &= u \\
\mathrm{wp}\,(v{:=}\,E)\,u &= \langle v \to u\rangle(E) \\
\mathrm{wp}\,(P\,.\,Q)\,u &= \mathrm{wp}\,P\,(\mathrm{wp}\,Q\,u) \\
\mathrm{wp}\,(\mathbf{if}\ c\ \mathbf{then}\ P\ \mathbf{else}\ Q)\,u &= (c \Rightarrow \mathrm{wp}\,P\,u) \wedge (\neg c \Rightarrow \mathrm{wp}\,Q\,u) \\
\mathrm{wp}\,(\mathbf{while}\ c\ \mathbf{do}\ P)\,u &= \mu w \cdot (c \Rightarrow \mathrm{wp}\,P\,w) \wedge (\neg c \Rightarrow u) \\
\mathrm{wp}\,(v\colon [\mathit{pre}, \mathit{post}])\,u &= \mathit{pre} \wedge \langle v_0 \to \forall v \cdot \mathit{post} \Rightarrow u\rangle(v)
\end{aligned}$$

Refinement can now be defined: P is refined by Q, written $P \sqsubseteq Q$, iff $\mathrm{wp}\,P\,u \Rightarrow \mathrm{wp}\,Q\,u$. Stepwise refinement is supported by this definition because it defines a partial order, and the program constructs are monotonic with respect to it; this immediately yields some obvious refinement laws, e.g., if we have $A \sqsubseteq A_1$ and $B \sqsubseteq B_1$, then we can conclude $A\,.\,B \sqsubseteq A_1\,.\,B_1$. Here are some less obvious refinement laws used later in this section.

1. Strengthened postcondition and weakened precondition:

   $$v\colon [\mathit{pre}, \mathit{post}] \sqsubseteq v\colon [w, u]$$

   if $\mathit{pre} \Rightarrow w$ and $u \Rightarrow \mathit{post}$.

2. Initialized loop:

   $$\begin{aligned}
   &v\colon [\mathit{pre}, I \wedge \neg c] \\
   \sqsubseteq\ &v\colon [\mathit{pre}, I]\,.\,\mathbf{while}\ c\ \mathbf{do}\ v\colon [c \wedge I, I \wedge 0 \leq V < \langle v \to V\rangle(v_0)]
   \end{aligned}$$

   if I does not contain initial values. Here, I is known as the loop invariant and V is known as the loop variant.

3. Assignment:

   $$v\colon [\mathit{pre}, \mathit{post}] \sqsubseteq v{:=}\,E$$

   if $v = v_0 \wedge \mathit{pre} \implies \langle v \to \mathit{post}\rangle(E)$.

4. Multiple assignments:

   $$v, x{:=}\,E, F[v/E] \sqsubseteq v{:=}\,E\,.\,x{:=}\,F$$

We now solve the maximum segment sum problem in this refinement calculus. Using the specification statement, the problem can be formalized as:

$$s, c, k : [\top, s = \mathit{outer}(\#L)]$$

Once again we define M by

$$M \;\widehat{=}\; s = outer(k) \wedge c = extra(k) \wedge 0 \leq k \leq \#L$$

Then we have these refinements:

$$\begin{array}{lll}
 & s, c, k : [\top, s = outer(\#L)] & \text{postcond.} \\
\sqsubseteq & s, c, k : [\top, M \wedge k = \#L] & \text{init. loop} \\
\sqsubseteq & s, c, k : [\top, M]\ . & \\
 & \textbf{while } k = \#L \textbf{ do} & \\
 & \quad s, c, k\colon [k \neq \#L \wedge M, M \wedge 0 \leq \#L - k < \#L - k_0] &
\end{array}$$

$$\begin{array}{lll}
 & s, c, k : [\top, M] & \text{assign., (6), (7)} \\
\sqsubseteq & s, c, k := 0, 0, 0 & \text{mult. assign.} \\
\sqsubseteq & s := 0\ .\ c := 0\ .\ k := 0 &
\end{array}$$

$$\begin{array}{lll}
 & s, c, k : [k \neq \#L \wedge M, M \wedge 0 \leq \#L - k < \#L - k_0] & \text{see below} \\
\sqsubseteq & s, c, k := s \uparrow (c + L(k)) \uparrow 0, (c + L(k)) \uparrow 0, k + 1 & \text{mult. assign.} \\
\sqsubseteq & c := (c + L(k)) \uparrow 0\ .\ s := s \uparrow c\ .\ k := k + 1 &
\end{array}$$

The second last refinement requires proving the following:

$$\begin{array}{ll}
 & s = s_0 \wedge c = c_0 \wedge k = k_0 \wedge k \neq \#L \wedge M \\
\Rightarrow & \quad \langle s, c, k \rightarrow M \rangle (s \uparrow (c + L(k)) \uparrow 0, (c + L(k)) \uparrow 0, k + 1) \\
 & \wedge\ 0 \leq \#L - (k + 1) < \#L - k_0
\end{array}$$

i.e.,

$$\left(\begin{array}{l}
\quad s = s_0 \wedge c = c_0 \wedge k = k_0 \\
\wedge\ s = outer(k) \\
\wedge\ c = extra(k) \\
\wedge\ 0 \leq k < \#L
\end{array} \right) \Rightarrow \left(\begin{array}{l}
\quad s \uparrow (c + L(k)) \uparrow 0 = outer(k + 1) \\
\wedge\ (c + L(k)) \uparrow 0 = extra(k + 1) \\
\wedge\ 0 \leq \#L - (k + 1) < \#L - k_0
\end{array} \right)$$

which is an application of simple arithmetic and equations (4) and (5). In fact, this is essentially the same proof obligation as (13).

5 Discussion

This section compares and discusses the differences of the two refinement calculi described in Section 4 in the following aspects: proving termination, expressing specifications that involve both pre-values and post-values, and choosing between looping and recursion.

5.1 Termination

In the predicate semantics, termination is replaced by timing because the resulting theory has less clutter, e.g., refinement is simpler and sequential composition is associative. It also has the effect of replacing proofs using induction and well-founded relations by those using arithmetic, even when programs are written with recursion, and therefore has a smoother learning curve. (An induction proof is implicit here: the base case hides in the initialization of variables, and the induction step hides in the loop.) Timing is also applicable to reactive systems, where a program is expected to never terminate and to produce output on time. The same approach generalizes to memory accounting and any accounting at all [16], permitting one to specify and verify bounds on resource usage in addition to time.

The predicate transformer semantics employs variants and well-founded relations in termination proofs. A terminating loop with such a variant can be shown to be a data refinement of a simple decreasing loop

$$\textbf{while } v > 0 \textbf{ do } v' < v$$

where v is a variable of a well-founded type and 0 stands for a minimal element of this type. (This is work in progress by Eric C. R. Hehner, Andrew M. Gravell, and Michael J. Butler.) Thus, the use of variants just reduces a termination problem to a simpler one, and we just *axiomatize*, in a sense, that the latter loop terminates. This assertive treatment of termination allows the definitions of refinement and sequential composition to stay symmetric and relatively simple (whereas the same attempt in the predicate semantics led to cumbersome definitions and laws).

A highly visible difference between the two approaches is refutability of claims of termination. Suppose you buy a program from me and I promise that it will terminate, but I break the promise. How much time must you wait or how much reasoning about the program (if I ever let you examine it) must you perform before you have enough evidence to make a case that I break the promise? Now suppose I promise that my program will terminate in one month; then after a month you will have a case. (If I only guarantee that it terminates in a century, you can choose not to buy it at all.)

In fact, in engineering practice, *good* specifications should bound resource usage of all sorts, including time and memory. The specification of the C++ Standard Template Library is a good start in this direction: appending an element to an expandable array is specified to take $O(1)$ amortized time, for example. An important activity in engineering is comparing components that have the same functionality but different performance trade-offs. If software construction is to become an engineering practice, software specification must come with sufficient information to support such comparisons.

Amusingly, for the reason of enforcible promises, Dijkstra argued against fairness (i.e., finitely often)[3], and then Lamport wrote a parody[4] and later a formal paper[5] pointing out that fairness is as enforcible a promise as termination, and therefore if one accepts one as a useful concept, one must accept both. Apparently, Dijkstra accepted termination but refused fairness, Lamport accepts both, and Hehner would refuse both.

5.2 Pre-Values and Post-Values

The predicate transformer semantics started by writing a specification as a precondition and a postcondition. This leads to some difficulty in expressing specifications like $s{:=}\, s + 1$. The theory is then made complicated to remedy this by introducing a "logical constant" X, writing $s\colon [s = X, s = X + 1]$. (The notation $s\colon [\top, s = s_0 + 1]$, which mentions pre-values in the postcondition, is defined as the equivalent specification that uses a logical constant.) In the presence of logical constants, the weakest precondition semantics of the specification statement becomes more complicated.

The predicate semantics takes a relational view to obtain a simple treatment: a specification is one single predicate about the pre-values and the post-values. This extends naturally to any kind of observation (communication, for example): a specification is a predicate mentioning all the observables.

5.3 Looping vs. Recursion

The predicate semantics emphasizes recursion, whereas the predicate transformer semantics emphasizes loops. This seems to be just a result of personal preferences, and both semantics can support both constructs equally well.

Loops are more common (in programming languages, education, and use), but the design of a loop is a lot of invention done at once: one must invent an invariant, a variant, and a condition, all at the same time. In contrast, the design of a recursion allows one to focus on one aspect of the three at a time.

Recursion also seems to be more flexible. A recursion involved in summing a list may be specified as

$$s = \Sigma L[0,..k] \implies s' = \Sigma L[0,..\#L]$$

or as

$$s' = s + \Sigma L[k,..\#L]$$

so one may use whichever one of the two that comes to the mind first. A loop invariant expresses the antecedent of the former only, so if one conceives the latter first, one must deduce the former. Imperative programmers tend

[3] EWD1013 Position Paper on "Fairness",
`http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1013.PDF`

[4] "EWD1013 Position Paper on Termination",
`http://www.research.compaq.com/SRC/personal/lamport/pubs/lamport-ewd1013.txt`

[5] with Fred Schneider, Another Position Paper on Fairness, *Software Engineering Notes* 13(3):1–2, July 1988.

to conceive the former because they describe what their state variable stores (s stores the partial sum); functional programmers tend to conceive the latter because they describe what their function achieves (`fold (+)` adds list elements to a seed value).

6 Current Questions

6.1 Extension to Object-Oriented Programming

Several recent works extend existing refinement calculi to object-oriented programming (see section 2.3), giving definitions and sufficient conditions of refinement of object classes. There are problems with these current works: some seem difficult to understand or use, some do not follow the principle of encapsulation, and some require the assumption that objects are immutable. Class refinement looks difficult, and whether this is because object-oriented programming is intrinsically convoluted or we just have not found the right theory yet is not obvious.

6.2 Theoretical Comparisons of Refinement Calculi

There are several predicate and relational semantics and corresponding refinement calculi in addition to the one in section 4.1, and they differ in treatments of termination or time. For example, while [14] uses a number variable to stand for time, there is a version that uses a boolean variable to stand for "finish". A question is then on the consequences of these differences.

6.3 Ease of Use and Tool Support

Refinement calculi are not widely used by programmers. A frequently cited reason is that little tool support has been available. An interesting question is then why programmers do not use refinement calculi, and what kind of tool support (and even education) is necessary to change this. One beginning work is my M. Sc. Thesis, *A Formal Tool for a Formal Refinement Method*. (Other beginning works are cited in it.)